AFRL-RI-RS-TR-2012-087

MELD: A LOGICAL APPROACH TO DISTRIBUTED AND PARALLEL PROGRAMMING

CARNEGIE MELLON UNIVERSITY

MARCH 2012

FINALTECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2012-087 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
CHRISTOPHER J. FLYNN
Work Unit Manager

/s/
PAUL ANTONIK, Technical Advisor
Computing & Communications Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAR 2012	Final Technical Report	JUL 2010 – SEP 2011

4. TITLE AND SUBTITLE

MELD: A LOGICAL APPROACH TO DISTRIBUTED AND PARALLEL PROGRAMMING

5a. CONTRACT NUMBER
FA8750-10-1-0215

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Seth Copen Goldstein
Flavio Cruz

5d. PROJECT NUMBER
BI20

5e. TASK NUMBER
CM

5f. WORK UNIT NUMBER
U0

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Carnegie Mellon University
5000 Forbes Ave
Pittsburg, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2012-087

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2012-1125
Date Cleared: 2 MARCH 2012

13. SUPPLEMENTARY NOTES

14. ABSTRACT

With the industry-wide shift to multi-core processors and the rise of large-scale clusters as the engines for web services, it is clear that the future is all about exploiting parallelism. Unfortunately, writing parallel or concurrent software has long been a notoriously difficult task for programmers. In particular, the goals of maintaining correctness while also achieving high performance are far more challenging in a parallel or concurrent environment compared with a single-threaded system. Meld is a logic-based programming language that has proven effective for programming massively distributed systems. In this report we describe Meld and how it can also be used to program more traditional parallel machines. We evaluate Meld on a small set of benchmarks and describe some of the methods we use to make it an effective tool for parallel programming. While more work needs to be done it is clear that the use of a single language for describing both computation and coordination can lead to clear, concise, and efficient implementations.

15. SUBJECT TERMS

parallel programming, logic programming, Meld

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
					CHRISTOPHER J. FLYNN
a. REPORT	b. ABSTRACT	c. THIS PAGE	UU	36	19b. TELEPHONE NUMBER (*Include area code*)
U	U	U			N/A

TABLE OF CONTENTS

LIST OF FIGURES

1 ABSTRACT

With the industry-wide shift to multi-core processors and the rise of large-scale clusters as the engines for web services, it is clear that the future is all about exploiting parallelism. Unfortunately, writing parallel or concurrent software has long been a notoriously difficult task for programmers. In particular, the goals of maintaining *correctness* while also achieving *high performance* are far more challenging in a parallel or concurrent environment compared with a single-threaded system.

Meld is a logic-based programming language that has proven effective for programming massively distributed systems. In this report we describe Meld and how it can also be used to program more traditional parallel machines. We evaluate Meld on a small set of benchmarks and describe some of the methods we use to make it an effectiv tool for parallel programming. While more work needs to be done it is clear that the use of a single language for describing both computation and coordination can lead to clear, concise, and efficient implementations.

2 INTRODUCTION

With the industry-wide shift to multi-core processors and the rise of large-scale clusters as the engines for web services, it is clear that the future is all about exploiting parallelism. Unfortunately, writing parallel or concurrent software has long been a notoriously difficult task for programmers. In particular, the goals of maintaining *correctness* while also achieving *high performance* are far more challenging in a parallel or concurrent environment compared with a single-threaded system. Maintaining the correct functionality of the software is complicated in part because a key to avoiding performance bottlenecks is to distribute (and often replicate) program state across the machine, and managing the concurrent accesses to this distributed state (while avoiding undesirable forms of race conditions or inconsistency) can greatly increase programming complexity. Compared with sequential systems, parallel and concurrent systems are susceptible to additional performance pitfalls including load imbalance, synchronization stalls, communication overhead, network bottlenecks, and, in cases with a shared address space (e.g., multi-core), more complex interactions with the memory hierarchy (e.g., false sharing). Assuming that the programmer has identified a sufficient source of fundamental parallelism, the key to achieving good performance typically boils down to *data partitioning* and *scheduling*: i.e., distributing the tasks (and their associated data) across the machine and prioritizing them in the right order such that the amount of work is evenly balanced while exploiting locality to minimize communication, minimize synchronization, and take full advantage of the memory hierarchy. This project investigated a new, general approach to let programmers solve both the data partitioning and the scheduling problem succinctly, provably, and all within a single language, Meld.

In the context of the Claytronics project [1], a massive distributed system with millions of modular robots [2], a new language called Meld has emerged. Developed by Ashley-Rollman et. al [3], Meld makes programming such types of systems possible, since it allows the *ensemble* to be seen as a unified whole. The language handles issues such as communication and synchronization automatically by compiling programs into fully

distributed code.

Meld is a bottom-up logic programming language based on Datalog [4]. In Meld, each fact must refer to a node in the ensemble where the fact will be stored and the communication between nodes is made implicit through the use of *structural facts*. This declarative approach removes the need to understand how the system manages state and how communication is implemented underneath, making programs far easier to write and read. Moreover, the use of logic programming makes programs particularly suited for correctness proofs.

This report describes our experiments in extending Meld to multicore and traditional distributed systems. While in ensembles the system is viewed as a graph of processing units, in traditional architectures we view it as a data structure that is manipulated by several threads or processes. For instance, in neural networks we have a graph of nodes, where nodes naturally correspond to neurons and each node contains several facts such as its layer type and the weight of its connections.

Previous results for ensembles showed that it is possible to write efficient and scalable programs on systems with over one million nodes [3]. For parallel and distributed architectures, we have applied Meld in machine learning algorithms such as loopy belief propagation, graph algorithms and sparse matrix problems. We developed a new compiler and runtime system that runs on multicore processors using Pthreads and also on distributed systems, using OpenMPI [5]. The compiler leverages information about the network topology to speed up programs, and the runtime system uses several strategies for manipulating the graph of nodes. Preliminary results indicate that Meld is able to scale programs in these architectures with minimal effort from the programmer.

2.1 Approaches to Parallel Programming

Over the years, researchers have explored a rich set of parallel programming models with varying degrees of success. At one end of the spectrum are lower-level programming abstractions such as *message passing* (e.g., MPI) and *shared memory* (e.g., OpenMP). An advantage of these lower-level models is *expressiveness*: with sufficient effort, one can control nearly every aspect of how the parallel program behaves (including data partitioning and scheduling), and one can apply these models to almost any appli- cation domain. The key disadvantage, however, is *programming complexity*: because programmers must explicitly manage asynchronous threads and worry about all of the possible ways in which their arbitrary interleavings could accidentally corrupt program state, most mortal programmers struggle mightily with correctness issues. In theory, an MPI (or OpenMP) wizard can squeeze every last drop of performance out of a machine because they can control everything, but in reality even programming wizards struggle with correctness and performance issues in these models once the applications become sufficiently large and sophisticated. The result tends to be a tight intertwining of the algorithm and the instructions for coordinating the parallel execution, making the program even harder to understand and maintain. In addition to these *scalability* challenges, *fault tolerance* is also left as an "exercise for the reader" in these lower-level programming abstractions.

To help address these correctness challenges, a number of researchers have explored *functional* (e.g., Id90 [6], [7], Haskell [8], and NESL [9]) languages as a way to create parallelism while reducing the amount of state that the programmer must explicitly reason about. These declarative and functional approaches help to simplify the programming task and

eliminate race conditions on data, but unfortunately they do not allow the programmer to control scheduling (hence performance can often be disappointing), and proof capabilities have not been demonstrated.

We observe that some of the most successful parallel programming models have been tailored to application domains where the models can provide a reasonably high level of abstraction (e.g., without diving into the gory details of how to manage asynchronous threads) while still providing a clear and sufficiently expressive model for data partitioning and scheduling. For example, synchronous dataflow (e.g., SISAL [10]) and stream-based languages (e.g., StreamIt [11]) have also been quite successful for streaming-type applications due to their simple *producer/consumer* data sharing model and their simple *pipelined* scheduling model. In the scientific computing domain, HPF's *data parallel* model [12] has been quite successful for dense matrix computations: rather than explicitly creating threads, the programmer simply specifies how the data is to be distributed across the machine, and the computation is scheduled accordingly using an *owner computes* model. In the arena of databases, SQL [13] is an example of a declarative language which works in a very limited domain, but is extremely successful.

A recent success story is the *MapReduce* programming model, which can be viewed as a somewhat more generalized version of the data parallel model that is optimized for large scale clusters. In MapReduce, the data sharing and scheduling model is very simple: the computation for the *map* phase is executed on the nodes that contain that data, the communication occurs only during the subsequent global *reduce* phase. The DryadLINQ [14], [15], Pregel [16], and Pig Latin [17] are programming models which are also designed for clusters and offer somewhat more general programming models than MapReduce, but they do not allow the programmer to specify scheduling strategies or support formal proof techniques.

Hellerstein's group has been working on a family of Datalog [18] descendants—P2 [19], Overlog [20], and SNLog [21]—that provided the initial inspiration for our own work and were designed for program- ming overlay networks and sensor networks. In contrast, Meld has added—among other things—support for robot actuation and sensing, Meld is based upon a well-defined formal semantics, and Meld has been shown to be amenable to formal proof. In principle, even very large ensembles can be reasoned about formally with Meld.

One clear theme that comes from looking at this impressive body of work is that there lacks a clear and concise method for coordinating the data partitioning and scheduling of threads separately from the main algorithm. The idea of a coordination specification, however, is not new. Early work in the context of Fortran showed that separating out coordination provided significant leverage for dense matrix codes with potentially unbalanced work [22]. Other work from this period includes Linda [23], CLP [24], and Schedule [25] which are all examples of languages which directly supported coordination. Our approach borrows ideas from this early work, applying them to Meld in the context of modern architectures.

In short, no other approach has been shown to create expressive, concise parallel programs which are scalable, amenable to proof, fault tolerant, and support an easy path towards efficiency by allowing programmers to optionally and incrementally coordinate the parallel execution of their program.

2.2 Our Approach with the Meld Language: from Ensembles to Parallel Systems

Our approach to parallel programming originated with our work on *claytronics* [26], [27]: a form of programmable matter that is an example of a massively distributed cyber-physical system. Because a claytronics ensemble might consist of millions of processing nodes which are constantly moving around each other and frequently experiencing failures, devising successful strategies for programming such an extreme system—where scalability and fault tolerance are so important—has given us a very different perspective on the parallel programming problem compared with most previous work. We believe that this unique perspective is our "secret weapon" in successfully attacking the parallel programming problem for a broad range of concurrent systems.

To understand where we started and how our insights might be more broadly applicable, we begin by dividing concurrent systems into two broad categories (each with two subcategories):

1. **Ensembles**:
   - Embedded-physical distributed systems (e.g., claytronics, sensor networks)
   - Geographically distributed systems (e.g., the power grid, the Internet)
2. **Parallel Systems**:
   - Cloud computing systems (e.g., Google-style clusters, etc.)
   - Parallel machines (centralized, synchronized, big-iron and multi-core)

The apparent differences between these systems have led to significantly different approaches to solving the notoriously hard problem of programming them. Traditional proposed solutions have focused on specific features of each type of system. Recent advances in the underlying hardware, however, have reduced the differences between these systems from the programmer's standpoint, making robustness, fault tolerance, and scalability essential for all of these systems.

To successfully program claytronics ensembles, we developed a new programming language that we call *Meld* [3], [28]. Meld is an extension of Datalog [46], [18] which uses a declarative approach to concurrent programming that is based upon logic programming. Meld's conciseness helps to limit the complexity burden on the programmer while writing sophisticated applications [28]. In the context of claytronics, we have demonstrated not only that Meld programs are scalable, efficient and fault-tolerant, but that they are also amenable to formal proof techniques [3], [29], [30]. As we will describe in detail in the remainder of this report, our success in programming claytronics ensembles can be transferred both to geographically-distributed ensembles (e.g., the power grid) and to large *parallel systems* (including multi-core architectures and large clusters).

2.3 Goal of Our Research and Expected Impact

One can think of MPI and MapReduce as being two different extremes along a spectrum of languages: MPI offers full expressiveness and control over data partitioning and scheduling, but at the expense of significant programming complexity; MapReduce simplifies the programming task and implicitly supports scalability and fault tolerance, but with limited application scope and no mechanism for programmers to specify data partitioning or scheduling strategies beyond the default.

Our hypothesis is that Meld sits at a sweet spot between these two extremes: Meld offers the pro- gramming simplicity of MapReduce (with scalability and fault tolerance baked in), but (1) has a broader application scope (e.g., it can efficiently handle highly irregular forms of

parallelism), (2) gives program- mers the ability to express clever data partitioning and scheduling strategies within the language itself, and (3) supports formal proof techniques.

We believe that the key to success for Meld is the fact that programmers can choose to *incrementally* specify the coordination of data partitioning and scheduling *in the same language* in which they are implementing the application itself. In the case of claytronics, the data partitioning and scheduling issues were naturally dictated by the physical nature and functionality of the ensemble: i.e., the data representing each node lived on that node, and the associated computation was scheduled on that node as well. For parallel systems, however, we need a more flexible approach. Our insight is to use *concepts* from coordination languages [22], [23] to express the partitioning and scheduling separately from the main program, thereby allowing programmers to only concern themselves with these details if they matter, with the important difference that we will use Meld for both the main program and its coordination. By using the same language for both the algorithm and the coordination, we believe that this will lead to a more expressive language as well as a more complete system which will be amenable to automatic proof.

To test our above-stated hypothesis we extended Meld to support these partitioning and scheduling directives, and evaluated Meld on several kernels—all pairs shortest path, pagerank, back-propogation in neural networks, a power distribution algorithm, and loopy belief propogation. We have investigated several scheduling methods, explored diffferent approaches to programmer controlled coordination, and for the first time effectively combined linear and classic logic to allow the programmer a controlled way to optimize their usage of memory. Now that the basic implementation has been performed and validated, the initial idea that Meld can be an effective tool for parallel programming, the next step in this research is to re-implement the coordination primitives that we initially experimented with. Combined with other work on proving Meld programs correct we believe that Meld can be an effective tool for programming parallel graph-oriented algorithms.

3 METHODS, ASSUMPTIONS , AND PROCEDURES

3.1 The Meld Language

Meld is a bottom-up logic programming language based on Datalog. Like Datalog, its execution consists of a database of *facts* plus a set of *production rules* for generating new facts. Each fact is an association between a *predicate* and a tuple of values. A predicate can be seen as a relation or table in which the tuples are its elements. Production rules have the form `p :- q1, q2, ..., qn.` , logically meaning that ”if `q1` and `q2` and ... and `qn` are true then `p` is true”.

3.1.1 Kinds of Facts

When a Meld program begins execution, the database is populated with axiomatic facts and with *sensing facts*, derived from observations about the world. As a Meld program executes, new facts are instantiated and added to the database. These derived facts represent program state and are called *computation facts*. They are used internally by the program to do computations and generate results.

Eventually, *action facts* are also derived. Syntactically, action facts are no different from

other facts, but semantically they cause some behavior to occur outside of Meld. In the world of modular robotics, this behavior is typically the locomotion of a robot from one location to another. In distributed and parallel systems action facts can be used to provide an output (to files or a terminal), interact with external functions, affect load balancing, etc. Note that action facts are not added to the database.

3.1.2 Rules

A Meld program contains a set of rules. A rule has two main components: the *head*, which indicates the facts to be generated; and the *body*, which contains the pre-requisites to generate the head facts. The body may contain the following as pre-requisites: *subgoals*, expression constraints or/and variable assignments. The head can only contain subgoals. Variables are limited to the scope of the rule and must be defined in the body subgoals or through the body variable assignments. Variables that only appear on the head are forbidden, since each instantiated fact must be *grounded*, i.e., its arguments must be instantiated. The abstract syntax for the language is presented in Figure 1.

Known Facts	Γ ::=	$\cdot k\Gamma, f(\hat{t})$
Accumulated Actions	Ψ ::=	$\cdot k\Psi, a(\hat{t})$
Set of Rules	Σ ::=	$\cdot k\Sigma, R$
Actions	A ::=	$a(\hat{x})$
Facts	F ::=	$f(\hat{x})$
Constraints	C ::=	$c(\hat{x})$
Expression	E ::=	$E \wedge EkF k \forall.EkC$
Rule	R ::=	$E \Rightarrow FkE \Rightarrow Akagg(F, g, y) \Rightarrow F$

Figure 1: Abstract syntax for Meld programs

Whenever all body pre-requisites are satisfied, the head subgoals are instantiated as facts and then they are added to the program database (except when action facts are derived). To satisfy all pre-requisites, the body subgoals must be matched against the program database. Both constraints and subgoals match successfully when a consistent substitution is found for the body's variables such that one or more facts in the database are matched. Constraint expressions are boolean expressions that use variables from subgoals (and thus database facts) and from variable assignments. Allowed expressions in constraints include arithmetic, logical and set-based operations.

Each predicate used in the program must be explicitly typed. Each field is either of a basic type or a structured type. Basic types include the integer type, floating point numbers and the *node address* (see 3.2.1). A structured type includes lists of basic types.

```
type virtual neighbor edge(node, node, int).
type path(node, node, min int, list node).

path(A, A, 0, [A]).

path(A, B, D + W, [A | P]) :-
edge(A, C, W),
path(C, B, D, P).

write(A, B, D, P) :-
terminated(A),
path(A, B, D, P).
```

Figure 2: All-Pairs Shortest Path algorithm.

3.1.3 Evaluation

Meld programs are run in a forward-chaining or bottom-up style. This means that we consider the set of rules and the current database and look for rules that can be applied to generate new facts or actions, updating the database or performing the right actions. We repeat this process until we reach *quiescence*, meaning that no additional facts or actions can be derived.

3.1.4 Aggregates

In contrast to Datalog, Meld does not have negation, but has *set aggregates* [47]. The purpose of an aggregate is to define a type of predicate that combines the facts of that predicate into a single fact. The definition of an aggregate includes the field of the predicate and the type of operation to be done on the collection of facts.

Consider a predicate $p(f_1, ..., f_{m-1}, \texttt{agg}\ f_m, f_{m+1}, ..., f_{m+n})$, where f_m is the aggregate field associated with the operation `agg`. For each combination of values $f_1, ..., f_{m-1}$ there will be a set of facts matching the same combination. To calculate the aggregated fact of each collection, we take all the f_m fields and apply the `agg` operation. The fields between f_{m+1} and f_{m+n} for the aggregated fact are chosen depending on the operation.

Meld allows several aggregate operations, namely: max, to compute the maximum value; min, to compute the minimum value; sum, to sum all the values of the corresponding fields; and first, to non-deterministically select the first fact.

Figure 2 presents the All-Pairs Shortest Path algorithm. We use an aggregate to select the path with the minimum distance between two pairs of nodes. The minimum path is also built as a list. Since the list argument is on the right of the aggregated field, this list is taken from the `path` fact with the minimum distance.

3.1.5 Deletion

When facts in the database become stale (are no longer true) they are removed from the database. This may happen if some sensing facts about the world are no longer true or if the wrong aggregated fact was computed (Section 3.3.1 further explains this). Doing deletion

causes all facts derived from them to become stale and also be removed from the database. This process continues until all stale facts have been removed and the database consists solely of true facts. The mechanisms for doing this are subtle and are explained in detail by Ashley-Rollman et al. [3].

3.1.6 Linear Logic

One of the issues in the current Meld design is the problem of state. We have been able to solve some of those problems by detecting sets of rules with a *state argument*, that is, rules where the first argument of each predicate corresponds to the current iteration level. For example, all the programs in the previous section, except the All-Pairs, perform iterative refinements by indexing facts by the iteration number. Inspired by XY-stratification [31], we detect XY cliques and then insert delete operations to delete facts in older iterations, thus reducing the memory usage of such programs.

While the previous method solves memory issues, it is still awkward to express stateful programs in Meld. We are currently investigating how to model state directly in the language by leaving the limitations of classical logic and moving to a more expressive logical foundation. Our current direction is to use linear logic [32]. By using the resource interpretation of linear logic in a distributed context, we hope to implement new classes of algorithms in a declarative high-level manner. A (non-distributed) example of a logic programming language with a bottom-up semantics is provided by LolliMon [33]. LolliMon also integrates backward chaining for local, deterministic programs which may also make sense in our new context.

3.2 Distribution and Optimization

In this subsection we describe several code analyses and transformations performed by our compiler that turns Meld programs into optimized code able to run in a distributed environment. We first describe localization, a technique that allows the local evaluation of rules in the same place and automates messaging between workers. We then describe other techniques developed in order to reduce the use of deletion during execution, through the use of stratification and aggregate annotations. Finally, we explain an analysis technique to deal with memory problems in refinement algorithms.

3.2.1 Localization

Another difference between Meld and Datalog is that, in the former, the first field of each predicate must be typed as a node address. In rules, the first argument of each subgoal is called the *home variable* and refers to a *node*, the place where facts are stored. This convention originated in the context of declarative networking [34], namely, in the P2 system. For modular robotics, the node is a physical identity, the robot, an independently executing processing unit that, together with other nodes, form a graph node. For parallel and distributed architectures, each node corresponds to an element of the graph data structure.

While each fact must be associated with some node or data structure element, a rule may contain subgoals that refer to different nodes. In order to simplify the activation of production rules, each rule is transformed so that all subgoals refer to the same node. The process of *localization* is fundamental in the distribution of computation.

Before localization, each rule is either a *local rule* or a *link-restricted rule*. A local rule

does not need to be transformed since only facts from the same node are needed. A link-restricted rule is a rule where subgoals may refer to different nodes. This type of rule must be transformed so that the correct subparts of the rule are matched on the right nodes. This is enforced by matching all the subgoals in the node they refer to. To accomplish this, rules use *structural facts* that convey the graph topology of the nodes (i.e. node edges) and restrict how facts from different nodes can be used.

For an example, consider the code in Figure 4, where a rule refers to three different nodes, A, B and C. To localize this rule, we first build a tree representing the connection paths between nodes by picking an arbitrary root node, in this case, the node A, and then by adding edges using structural facts. The resulting tree is represented in Figure 4. Note how the constraints were placed in the bottom-most node where all the variables used in the constraint were available. This optimization aims to reduce communication by testing constraints as soon as possible.

Once the connection paths are known, localization transforms the original rule in Figure 3 into the *send rules* presented in Figure 5, that are characterized by having the same node in the body and a different

```
fact2(A, 2) :-
   fact1(A, A1), A1 > B1,
   edge(A, B), fact1(B, B1), B1 > C1,
   edge(A, D), fact1(D, D1),
   edge(B, C), fact1(C, C1).
```

Figure 3: Original code.

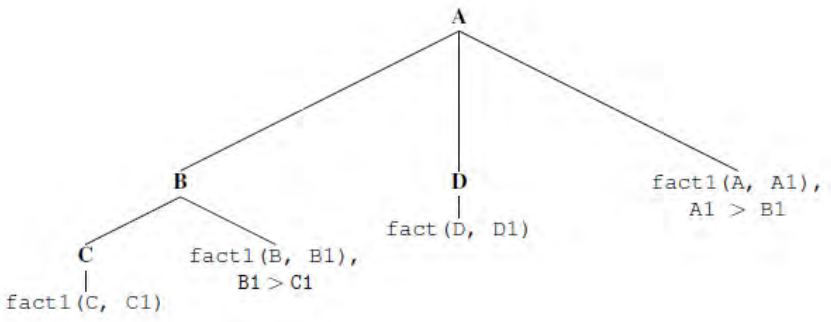


Figure 4: Localization tree for code in Figure 3.

node in the head. Once a send rule body matches, all the subgoals in the head are instantiated and then sent to the corresponding node, where the matching of the original rule can continue, until the complete rule matches. This is the basis for communication between nodes that makes Meld viable for distributed computation.

3.2.2 Local Aggregates

After the program is type-checked and localized, the compiler performs the *stratification phase*. In this phase, we compute the *local stratification levels* of the program (or *local stratums*), where the predicates that are computed using local rules are ordered. This way, structural facts appear on level 0, then the predicates that can be computed using the predicates in previous level on level 1, and so on, until all such predicates are stratified. Strata are separated by local aggregates, so that if several non-aggregate predicates are only used in local rules, they will form a single stratum. For predicates that are also aggregates, they serve as separators of strata, so that in stratum i+1, the aggregate is computed using all the facts computed in stratum i.

By using local stratification, we can generate *local aggregates* - aggregates that only need facts from the local node - by keeping track of the current stratification level. Our compiler recognizes such situations and is therefore able to pass information to the runtime system so that local aggregates can be generated safely once the previous local strata is fully computed.

Local aggregates are part of a class of aggregates called *safe aggregates*. For a safe aggregate, we know when it should be safely generated, since all the facts needed to compute it are available. Therefore, we do not need to employ deletion, because the aggregate will always have the correct value. This may not happen in the case of *unsafe aggregates*. For example, in the All-Pairs Shortest Path program in Figure 2, the min aggregate depends on facts coming from remote nodes, so `path` is not a local aggregate. Moreover, even if, for example, we waited for all the minimal paths to reach the target node, this may not work because the graph may have cycles which would lead to deadlocks between nodes.

3.2.3 Neighborhood Aggregates

In Meld, the graph structure where nodes are part of, is an important abstraction. In our experience, most aggregates that depend on remote facts are based around the neighborhood abstraction. Therefore, it is very usual to compute the sum or minimum of a set of facts that come through the input edges.

For these aggregates, if we have a fact for each input or output neighbor we can compute the aggregate safely. To this end, we augment the aggregate syntax to, optionally, include a neighborhood directive. For

```
__edge(X, Y) :- edge(Y, X).

fact2(A, 2) :-
    fact1(A, A1), A1 > B1,
    __remote1(A, B1), __remote3(A).

__remote1(A, B1) :-
    __edge(B, A), fact1(B, B1), B1 > C1,
    __remote2(B, C1).

__remote2(B, C1) :-
    __edge(C, B), fact1(C, C1).

__remote3(A) :-
    __edge(D, A), fact1(D, D1).
```

Figure 5: Localized rules for code in Figure 3.

example, consider the PageRank program in Figure 6, where the calcRank aggregate is computed as the sum of a fact from the home node and from all the input neighbors.

```
type virtual neighbor edge(node, node).
type pagerank(node, int, float).
type calcRank(node, int, sum float [<- edge]).
type numLinks(node, sum int).

const damping = 0.85.
const iterations = 35.
const pages = @world.

numLinks(A, 0).
numLinks(A, 1) :- edge(A, B).

pagerank(A, 0, 1.0 / float(pages)).
pagerank(A, I, V) :-
   I > 0,
   calcRank(A, I, T),
   V = damping + (1.0 - damping) * T,
   I <= iterations.

calcRank(A, I + 1, 0.0) :-
   pagerank(A, I, _).
calcRank(A, I + 1, O / float(C)) :-
   edge(B, A),
   pagerank(B, I, O),
   numLinks(B, C).
```

Figure 6: PageRank algorithm.

This new syntax slightly changes the semantic of the corresponding aggregate. For the case of the sum aggregate, it indicates that the calcRank aggregate is to be computed from the set of the first facts that generated by each neighbor. Our runtime system therefore discards any extra facts that may come from a neighbor.

Our compiler applies a source transformation to easily tag the source of each neighbor fact. First, it reads the direction of the arrow in the aggregate definition and looks up for the corresponding structural subgoal in the rule. It then adds an extra argument to the subgoal that corresponds to the source of the fact. The compiler also detects local rules and marks the source as the home node. For the PageRank program, the affected rules would be transformed to the new rules presented in Figure 7.

```
calcRank(A, I + 1, 0.0, A) :-
    pagerank(A, I, _).
calcRank(A, I + 1, O / float(C), B) :-
    edge(B, A),
    pagerank(B, I, O),
    numLinks(B, C).
```

Figure 7: Transformed rules using a neighborhood aggregate.

Neighborhood aggregates are also safe aggregates, since we can compute them safely when we receive all the needed facts from the neighborhood. Looking back at the shortest path program, we may also use neighborhood aggregates if the graph as described by structural facts has no cycles, thus effectively making the `path` aggregate a safe aggregate.

3.2.4 Detecting Iterative Refinements

While recursive rules were initially ignored in the field of deductive databases, much research has later been the done in the evaluation of such rules. This spawned several models such as the perfect model, well-founded semantics, stable model, etc. A particularly attractive model, named XY-stratification [31], restricts recursive rules by imposing that the first argument of each predicate, called the *stage argument*, must represent the stratification level. This maps naturally to logical algorithms that perform iterative refinement, like the PageRank program in Figure 6, where the rank of each page is improved from the previous rank. This class of programs is specially amenable to space optimizations.

Recursive rules form a recursive clique called the *XY clique*. Our compiler recognizes such cliques and ensures that any aggregate inside the XY clique is safe, so that there is no chance for deletion to happen. Since the program will always perform forward progress, we eliminate all facts from the previous iteration to improve memory usage.

For the PageRank program presented in Figure 6, the compiler detects a recursive clique containing predicates `pagerank` and `calcRank`. The predicate `calcRank` is an aggregate that depends on values coming from the neighbor nodes, so it is a safe aggregate. Our compiler introduces delete instructions in the body where `calcRank` and `pagerank` are consumed. Therefore, we are able to keep the memory usage constant, since we only need to store the current iteration.

From the body of all rules that generate `calcRank`, only one `calcRank` will come from each neighbor for each iteration and another is generated locally, so if we wait from a `calcRank` from each neighbor, the aggregate is generated safely. This kind of reasoning is easy to do since both `edge`, `pagerank` and `numLinks` is unique for each node, but for more complex problems it is more difficult to deduce this. We can thus easily delete the previous iterations as new iterations are computed because no deletion will happen. This analysis technique allows us to keep our language within its logic realm, while keeping it memory efficient when dealing with state.

3.2.5 Linear Logic

While state manipulation using classical logic is possible, it is much more easily accomplished using linear logic. In addition, having linear logic available makes implementing randomized and approximation algorithms much easier. In approximation

algorithms, we want an approximate result that is much faster to compute than the optimal result. Examples of such programs are the asynchronous PageRank [35] and certain classes of belief propagation algorithms such as SplashBP [36].

An example of a randomized algorithm is presented in Appendix A.5. This program solves the power grid problem, where we have a set of sinks and a set of sources and we want to connect each sink to a source in such a way that no source is overloaded. We first derive a linear fact `unplugged` for each sink and `load` fact for each source to mark the load as 0. Then, the third rule randomly picks a source for each sink, thus consuming the initial `unplugged` fact. The fourth rule selects an overloaded source and switches one of the connected sinks to a different source in order to reduce the source's load. This rule is applied several times and the program will converge to a state where the load is equally distributed. Finally, we use the `proved` sensing fact to detect if the `pluggedIn` fact has been derived enough times already (the program could run forever) and a `terminate` action fact may be generated to halt execution.

3.3 Implementing Meld on Parallel and Distributed Systems

Meld was originally implemented as an ensemble programming language, targeting modular robotic systems such as Claytronics [1]. These systems consist of many autonomous nodes passing messages to their physical neighbors. As such, the Meld implementation for ensembles described by Ashley-Rollman et al. [3] addresses many of the questions of how to implement Meld's language features on an ensemble. Here we focus on the implementation details unique to the parallel and distributed systems world. In addition to having different technologies to make use of, such as OpenMPI [5], we also have new challenges to address. In particular, a modular robotic system has a natural matching between computation and processors. Initial facts are stored upon the robot that sensed them. Similarly, derived facts have a natural matching to agents in the system. This distribution of data leaves little choice to be made in the division of computation to the various nodes. The parallel and distributed systems that we discuss here, have no natural matching of data and computation to processors, therefore we must devise new strategies that map nodes to processors.

Instead of viewing the distributed system as a graph of processing nodes, we view it as a graph data structure, where each data structure element (or node) stores computation facts as its attributes and uses them for computation. The data structure elements communicate with each other using send rules through the use of structural facts. Apart from computation and structural facts, we also consider sensing facts and action facts. Action facts are used as a means to write output to the terminal or to files. Sensing facts were used in modular robotics to make the robots sense the outside world, but here we use them to sense changes in the underlining execution model or to coordinate and balance the work load. For instance, we have the following sensing facts: `terminated`, to sense when the computation has finished; `colocated`, to determine if two nodes are in the same process; and `select_nodes` to distribute the nodes across workers. We are exploring which sensing facts may be useful as we apply Meld to new applications.

The All-Pairs Shortest Path program in Figure 2 is a good example program where all types of facts are used: `path` is a computation fact; `edge`, a structural fact; `terminated`, a sensing fact; and `write`, an action fact. The `terminated` fact allows the program to write the shortest paths to the terminal once all paths are known.

3.3.1 Execution Model

In our execution model, we have the graph of nodes as a data structure and a set of workers. A worker is a processing unit that is either a *thread* or a *process*. A worker is able to process facts for any node. The processing of a fact from a given node is thus the basic *unit of work* in our parallel system. We impose two restrictions on work allocation. First, a node may only be processed by at most one worker at a given point in time. Second, a worker may only process one node at the same time. This disallows the manipulation of a node by multiple workers.

Evaluation of bottom-up logic programs can be done using several techniques, one of the most well- known being the *semi-naïve fixpoint* evaluation [37], [38]. However, these techniques are more appropriate for a centralized evaluation and are expensive in a distributed environment. To this end, we use the *pipelined semi-naïve* evaluation from P2 [34], which relaxes the semi-naïve fixpoint by processing each new fact separately. When a fact is proven to be true, all rules that use the fact in their body are selected as *candidate rules*. For each candidate, the rest of its rule body is matched against the facts in the node, so that new facts can be proved. This forces workers to process new facts as soon as they are generated.

Since parallel programs are not as dynamic as ensemble programs, we aim to avoid the generation of aggregates from partial information. For programs with only safe aggregates, this is not an issue, however, for programs with unsafe aggregates we use deletion if the aggregated fact is invalidated later. This, however, may lead to a significant part of the computation being invalidated and recomputed. For example, in the All-Pairs Shortest Path program in Figure 2, minimal paths may be computed based on partial information and must be invalidated.

Due to the non-deterministic nature of parallel and distributed workers, the amount of invalid compu- tation is highly variable, which degrades performance since execution times may vary widely between executions. In order to reduce this problem, we introduce the notion of a *computation round*. Each round proceeds as follow:

- Workers process all regular facts and safe aggregates;
- When a worker has no more work left to do, it enters into the *idle state*;
- Once the graph reaches a quiescent state, that is, when all workers are in the idle state, workers synchronize using a *termination barrier*;
- Unsafe aggregates are then generated by all workers;
- If some aggregate was generated (or deletion is to be done), it is considered as new work and the next computation round will be started;
- If no aggregates were generated, the computation is marked as complete and then finishes.

Since a termination barrier, a global operation, is used between computation rounds, parallelization is likely improved when the number of rounds is minimized. Making the generation of aggregates safe is thus an important endeavor, since it eliminates the need for more than one computation round and thus increases asynchronicity between workers and the throughput of the whole system. In practice we have not ever had to use an aggregate that the compiler could not prove was unsafe and all the programs had no intermediate computation rounds.

3.3.2 Graph Clustering

The goal of the runtime system is to balance work between workers. We thus want to maximize throughput and parallel efficiency by using a specific strategy that maps workers to new facts. Throughput is maximized when workers can process more work without the need to communicate with other workers. One way to tackle this is to cluster closer nodes and partition the clusters between workers. Since communication between nodes of different clusters is reduced, communication between workers is also reduced, which increases locality in the computation.

We introduced the *node address constant* to make it possible to declare all the graph information in the source code. Our compiler is modified to deduce the number of nodes in the graph and to build its structure from structural axioms. Each address is prepended by the symbol @ (for example, edge(@1,@2)). During compilation, and after the parse and type-checking phases, the *graph clustering* phase is called. It builds an internal representation of the graph, by mapping each node address a to a normalized node address n.

Formally, this mapping is represented by the bijective function $M(x)$, where the domain is the set of nodes described in the program's source code and the codomain is the discrete interval $[0, N]$, where N

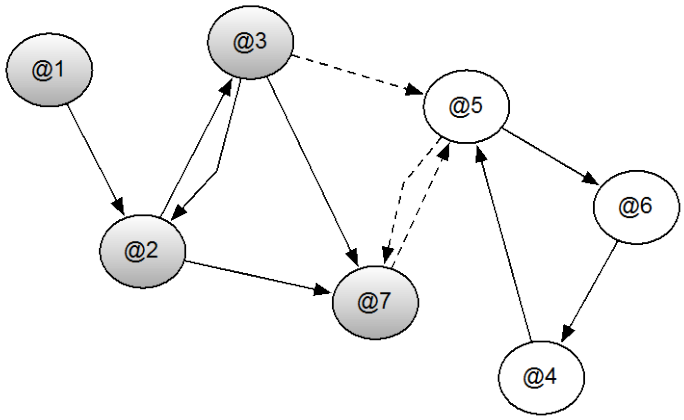


Figure 8: Clustering using a breadth-first method.

is the number of nodes in the graph. The byte code of a Meld program includes all the pairs $(x, M(x))$ so that the runtime system can put this information to use.

We have four methods for defining the function M, namely:

- *Randomized*: the mapping is done randomly.
- *Naive*: nodes are ordered as they appear in the source code.
- *Breadth-First*: the mapping is built by picking an arbitrary node, n_{zero}, and by defining $M(n_{zero}) = 0$.

  Next, we pick all neighbors of n_{zero} and define their mappings in increasing order. Recursively, we pick their neighbors in a breadth-first fashion, until all the nodes have been visited.
- *User Defined*: the mapping is done through a *worker axiom* called `select_nodes`, that accepts as argument the order of the nodes.

Using a static approach to work balancing, we can easily slice the codomain of function $M(x)$ to distribute the nodes across T workers. It is also trivial to know the worker of a node n by computing the following expression: $min(n/(N/T), N-1)$, where N is the number of nodes.

The breadth-first method is used with the intent of clustering closer nodes in an ordered fashion. While not optimal, this method is simple and very efficient. For example, consider the graph in Figure 8. The node addresses represented are the ones included in the source code. Using the breadth-first method and starting with node 1, we get the following order: 1, 2, 3, 7, 5, 6 and 4. If we use a static approach with 2 workers, worker 1 would get nodes 1, 2, 3 and 7 and worker 2 would get node 5, 6 and 4. With this division only 3 out of 11 paths of communication exist between worker 1 and worker 2.

While for some programs it makes sense to programmatically define a specific node ordering, automatic ordering is an important endeavor since it fully automatizes the ordering phase. In the future, we intend to explore other techniques such as the METIS partitioning method [39].

3.3.3 Multithreaded Execution

Our Virtual Machine (VM) supports multithreaded execution using Pthreads and has been developed to explore different scheduling strategies. Here, we outline the strategies to distribute computation between threads.

- Static Division (SD)

  In this strategy, the nodes are divided uniformly between threads and each thread has a queue of *active nodes*. The nodes themselves have queues of new facts. An active node is a node with some facts to process.

  To obtain work, a thread pops an active node out of its queue and then processes all the node facts until the node's queue is empty. In this case, the node is then marked as *inactive*. When a local or send rule is fired and the corresponding target node N is owned by the executing thread, the new fact is pushed into the node's queue. If N is currently inactive, it must be set as active and then reinserted into the queue of active nodes of the executing thread. If a send rule is fired and N is located on another thread T, the procedure is similar, except we must take extra care when reinserting the node into T's queue.

  Once a thread has no more active nodes to process, it enters into the idle state. Here, the thread tries to synchronize with the other threads in order to terminate the computation round. We adapted the termination barrier proposed by Flood et al. [40] to implement termination detection. It includes an atomic counter representing the number of active threads, a boolean flag for each thread representing the current thread state and a global sense flag that is activated when the counter reaches zero. This is used in all the multithreaded strategies.

- Dynamic Division With Work Stealing (DD)

  In **DD** each thread starts with a pre-defined set of nodes as in **SD**. However, the division of nodes may not remain static and nodes can move to other threads if threads start starving for work. When a thread has no more work, it enters into the idle state, whereupon the thread selects a random *target thread* to steal nodes from its queue.

  Each thread has a *set of nodes* and each node is augmented with a field called owner that points to the owner thread. The set of nodes is also used to generate unsafe aggregates during the final phase of each computation round.

- Dynamic Division without Ownership (DDWO)
  DDWO is a dynamic scheduler like **DD**, except that nodes are not owned. There is only one queue of active nodes from where threads can pop nodes. The advantage of this method is that there are no costs in stealing nodes. However, there may exist some contention when accessing the global queue. Several programs such as Neural Networks, where the flow of computation moves from different sets of nodes, can take advantage of this scheduling strategy.

3.3.4 Distributed Computation

Our VM uses the MPI subsystem to do distributed computation. For each MPI process, we may have several threads executing concurrently. There is a *leader thread* per process that works with other leader threads to detect termination of computation rounds.

The communication between threads now depends on where the other thread is running. If it is on the same process, all the rules presented before apply. If, however, they are in different processes, we must use *message passing*. Each thread can independently send messages to *remote threads*. New facts instantiated from send rules that must be sent to remote threads are not sent immediately. Instead, each thread has a data structure called the *message buffer* for each remote thread, where new facts are marshaled and buffered to be sent later. When the message buffer reaches a certain size (usually, the MPI message limit), all the facts in the buffer are sent using a single MPI operation.

Send operations are done asynchronously, so that threads can put the message on the MPI subsystem and then continue working, without waiting for the remote thread to receive the message. However, the memory used to marshal facts must be freed later, after the thread makes sure the messages were received. For this, each thread maintains several lists of *request handlers*. Each request handler is a pair with a memory pointer to a list of serialized facts and a MPI request object. Subsets of request handlers from the request list are then checked periodically to test if the send operation has been completed.

Facts that need to be sent to remote threads are indexed and sent directly to the corresponding remote thread. When receiving work, threads ask for their corresponding work from the MPI subsystem. When all threads in a process are idle, all threads synchronize and the leader thread communicates with other leader threads to detect termination. The detection of termination is done using the Dijkstra-Safras token-ring based algorithm [41].

While the division of nodes between threads in the same process is done using different strategies, the division between processes is done statically. We want to explore in the future if a more dynamic approach may be useful, however this may require the movement of nodes in the network, which may be too costly.

3.4 Implementation Details

3.4.1 Compiler

The compiler is a 3500 lines program written in Common Lisp. After reading the source code file, the compiler builds the Abstract Syntax Tree (AST) and then compiles it into a list of instructions, one list for each predicate. These instructions and the ordering of nodes are then written into the byte-code file.

3.4.2 Virtual Machine

The VM is implemented in C++ and makes use of the Standard Template Library and Boost.

We use the Boost.Threads package for multithreaded code and Boost.MPI as an abstraction to OpenMPI. To start a program, the VM reads the byte-code file passed as an argument and sets up the threaded environment, and optionally communicates with other remote processes to coordinate distributed execution. The machine architecture was designed with extensibility in mind: we have a base class for all schedulers that can be extended to implement new scheduling strategies. We next describe some interesting details of our machine.

- Lists
  Lists are represented as *cons cells*. We use *reference counting* to garbage collect unused lists with the use of an atomic integer for each cell.
- Queues
  We use lock-free data structures whenever possible. The queues used in **SD** only have one thread popping from the queue while having multiple pushers. These are lock-free queues where only the push operation needs to use *Compare-And-Set (CAS)* operations and the pop operation only needs to adjust the sentinel node (there is no race condition between the two operations). These queues are adapted from [42]. Since predicates are stratified, we use a binary tree bounded priority queue [43] for queues of facts, another lock-free data structure, where each level contains a lock-free queue. For the schedulers **DD** and **DDWO**, the queue of nodes uses locking since the queue has multiple workers pushing or popping nodes. The queue of facts for each node is the same as before, a lock-free binary tree bounded priority queue.
- Tries
  Facts are stored using the trie data structure. Tries are trees where facts are indexed by the common prefix. Our trie implementation was inspired by the work done in tabling for Prolog [44], where tries were used to store Prolog terms. The original design maintains each trie level as a simple linked list, which may be replaced by a hash table if this list gets too big in order to make search faster. In our implementation, we may also need to delete items from the trie, so we optimized the data structure to make deletion an efficient operation. Instead of simple linked lists, we use double linked lists and trie nodes may point to their hash table buckets so that we can delete the hash table in certain situations.
  Each node uses a trie per predicate to store facts. During evaluation of rule bodies, we build a *match object*, which matches some arguments of the target predicate to instantiated values, so that by searching the predicate trie from top to bottom, we can easily discard invalid trie branches. For a collection of facts that need to be aggregated, we use a separate trie for such facts. When we need to iterate over the collection, we use a double linked list of trie leaves (that is maintained by the trie operations) for direct access of each stored fact.
- Memory
  Memory allocation is done using pools of memory objects local to each thread. This reduces con- tention in memory allocation.
- Barriers
  During initialization we sometimes need to synchronize the threads. We use a Combining Tree Barrier [45] with a global sense flag for these situations.
  During each computation round, we need to assess if we should terminate or continue to the next round. We use a special barrier here, formed by a global atomic counter, a global triple-state flag and a local triple-state flag, one for each thread. After a thread

generates its aggregates, it decrements the global counter. If this is the leader thread, it waits for the counter to reach zero. The other threads wait for a change in the global flag (by comparing it to their local flag). The state of each flag can be 0, 1 or 2, shifting to 1 or 2, depending if the current round is even or odd. When the leader thread changes the global flag, the other threads read the result and change their flags accordingly. If the global state is set to zero, it means that the computation is over. For synchronization between processes, we use the `MPI_Allreduce` operation after all the threads in the process have synchronized.

- Initialization
  During initialization, each node needs to initialize its edge facts and axioms. To do this efficiently, we construct an hash table with N buckets for the `init` procedure of the byte-code, where N is the number of nodes. Each node reads its bucket and jumps directly to its specific code.

4 RESULTS AND DISCUSSION

We have written several algorithms to experiment with our compiler and runtime system. All of them use recursive rules with aggregates and, except for the All-Pairs Shortest Path program, they work by iterative refinement. The programs written are the following [48]:

- PageRank algorithm;
- All-Pairs Shortest Path algorithm;
- Neural network training using the back-propagation algorithm;
- A naive loopy belief propagation algorithm to de-noise images.

The All-Pairs Shortest Path algorithm (presented before in Figure 2) is prone to deletion and needs several computation rounds to complete because it has one unsafe aggregate. We used two different datasets: US 500 `Airports`, a graph of the 500 most important US airports with the corresponding weighted connections [49]; and `grid`, a configuration where the graph forms a square grid with cycles.

The Neural Network algorithm uses XY-stratification to order each training pattern and to generate improved neural link weights in subsequent iterations during the *back-propagation phase*. We used a neural network with 3 layers: input, hidden and output layers. We have 32 neurons in the input layer, 16 in the hidden layer and 16 in the output layer. The number of training examples is 500.

In the loopy de-noise algorithm we use a 100 by 100 grid configuration of pixels that correspond to the image we want to de-noise. We start with a single belief value for each pixel that is updated as iterations advance. Both the belief facts and the messages exchanged between the pixels are XY-stratified, so older values are deleted, and only the most recent belief and messages values are kept.

For the PageRank program, we use several graph configurations, namely: `grid`, where the pages form a square grid; `cycle`, where nodes are linked sequentially and the last node links to the first one; and a more realistic dataset of web links called `search engines` [50]. We compute the rank values by using 35 iterations.

4.1 Experimental Results

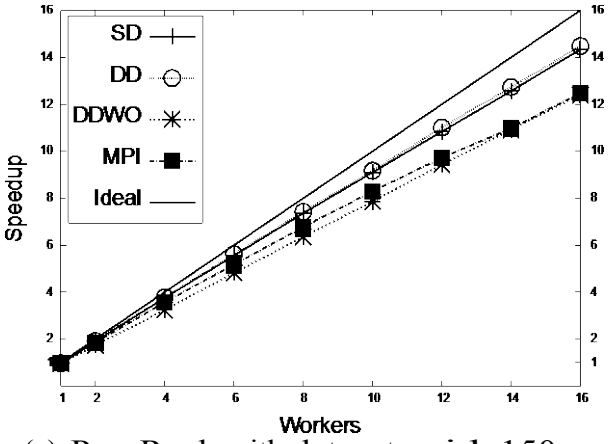
In our experimental setup, we used a machine with four AMD Six-Core Opteron TM 8425 HE (2100 MHz) chips (24 cores) and 64 GB of DDR-2 667MHz (16x4GB) RAM, running GNU/Linux (kernel 2.6.31.5- 127 64 bits). We compiled our virtual machine using GCC 4.4.5 (g++) with the flags -O0 `-std=c+0x -march=x86-64`. The libraries used were OpenMPI 1.4.3 and Boost 1.41. For MPI we executed all the processes on the same machine, so these results will be somewhat advantageous to this scheduler since the costs of network communication are nonexistent.

We developed a simple sequential scheduler that avoids the use of parallel and distributed synchro- nization mechanisms and is used as a comparison for all the other strategies. Note that for all the Meld programs, the nodes were ordered using the breadth-first approach.

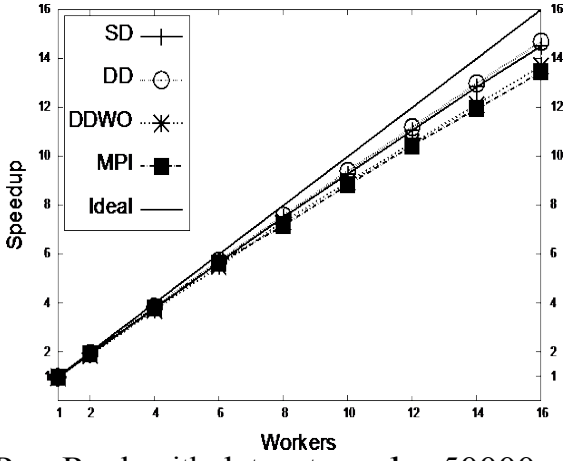
The results for the PageRank program are presented in Figure 9. Both the `grid` 150 and the `cycle` 50000 datasets are regular datasets, so we should expect similar results for all the schedulers, since most nodes have an equal number of connections and thus a static division should suffice to balance work. In these benchmarks, **DD** tends to perform slightly better than the other schedulers (except in the last benchmark) and **DDWO** performs worse than most schedulers, even against the static **MPI**. This is explained by the contention

when accessing the queue of active nodes. This rarely happens in **DD** since the work is well distributed initially, so there is little work stealing.

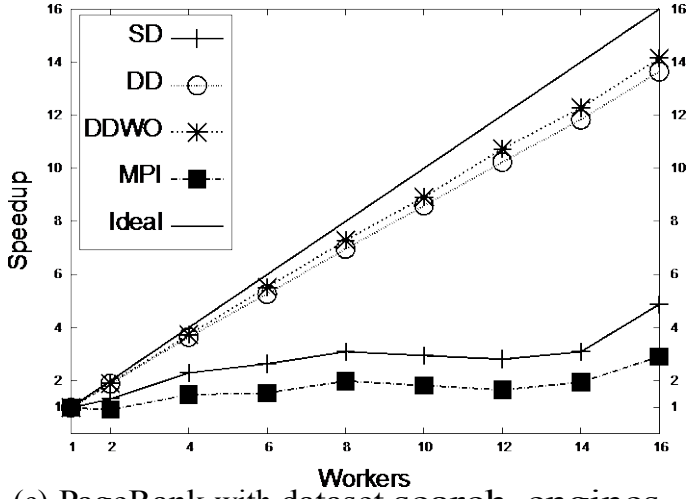
The `search engines` benchmark shows us how the PageRank program behaves in a more realistic graph. Notably, the dynamic schedulers all perform better than the static schedulers. As expected, **MPI** performs worse than **SD**, but its behavior is very similar to **SD**. We think this difference results from the extra work performed by **MPI** in serializing and then deserializing new facts.



(a) PageRank with dataset grid 150.



(b) PageRank with dataset cycle 50000.



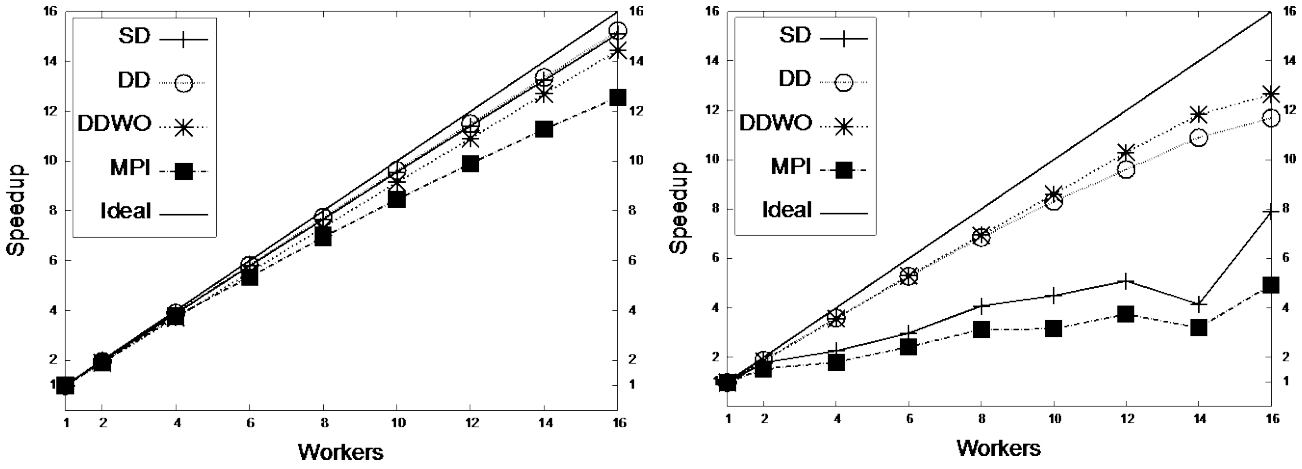
(c) PageRank with dataset search engines.

Figure 9: Performance results for the PageRank datasets. Note that both DD and DDWO come up on top and that MPI and DD usually trail each other, since they are both static schedulers.

In Figure 10 we present the speedups for the Loopy Belief Propagation (BP) and Neural Network (NN) program. The BP program is a very regular program: the graph structure forms a grid and the nodes simply exchange messages with its neighbors to improve their belief value. BP is thus a program where the work is easily balanced by just doing a static division. The evidence validates this assumption since all the schedulers perform equally well, even the static ones. In the speedup line, **DD** performs as well as **SD** for all the number of processors. Instrumentation of execution of the virtual machine indicate that little work stealing goes in **DD**, which explains why **SD** and **DD** perform identically.

The NN program is very different than the previous programs. For each training pattern, we start by computing the activations of the input, then the hidden and finally the output layer. Then, on a second round, we compute the deltas of each node, starting from the output layer to the hidden layer, and finally to the input layer. This pattern happens several times,

which makes it difficult to balance work across workers. This can easily be seen in the case of static schedulers, namely, **MPI** and **SD**, where the results are not as good as the other schedulers. Another important aspect relates to graph clustering. Notice that when the number of workers is 14, the speedup decreases suddenly, but increases a lot when the number of workers is 16. This effect is directly related to the graph partitioning used. In the last case, the nodes are evenly allocated more horizontally, which makes it more suitable for load balancing, since some layer



(a) Loopy Belief Propagation benchmark.

(b) Neural Network benchmark.

Figure 10: Performance results for the Neural Network (NN) and Belief Propagation (BP) programs. While BP is a program with a very regular and clear-cut computational pattern, in NN the communication and computation is not as regular, therefore it requires dynamic schedulers to perform well.

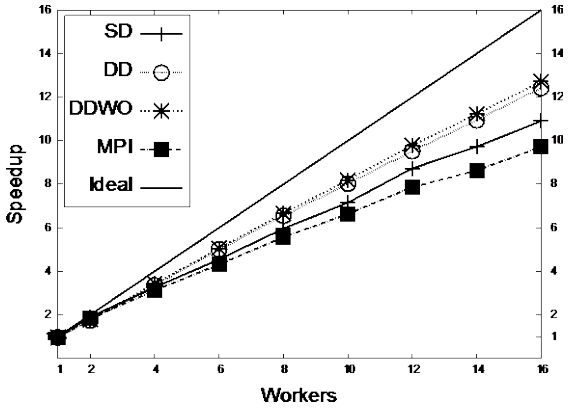
is doing some work at any given time. If a worker only gets nodes from the same layer, it will need to sit idle and wait for work if the computation is being performed on other layers. In relation to **DD** and **DDWO**, we observe that the latter usually beats the former. This happens because, as the number of workers increases, randomly selecting a thread to steal work from may not be as successful as using a global queue of nodes, since the target thread may be out of work.

In Figure 11 we present the results for the All Pairs Shortest Paths program. We experimented with two different versions of this program. In version *A*, we just compute the shortest distance and in the version *B*, we compute both the distance and the corresponding path as a list of nodes.

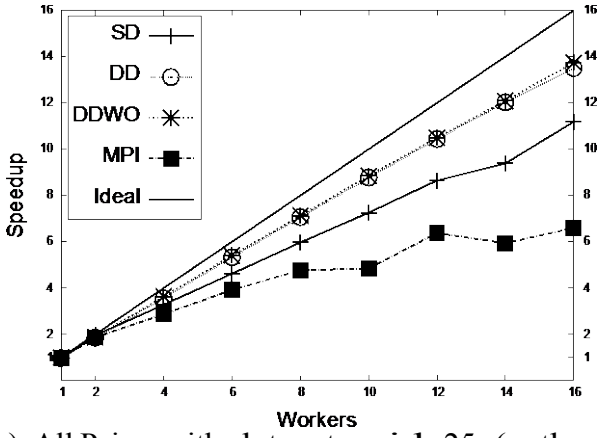
We experimented with the two versions with the `grid` dataset, Figure 11 (a) and (b), respectively. In both *A* and *B* versions, the threaded schedulers perform well, and the same scalability pattern can be observed. Moreover, note that in *B* the speedups decrease slightly, because we need to compute both the minimum distance and the corresponding path. This requires the construction of several lists and sublists during runtime that are eventually

shared between threads, thus reducing overall performance. Sublists references must also be maintained which requires more synchronization costs between threads. For the **MPI** scheduler, the slowdown is more noticeable, but here we need to serialize and deserialize the lists, which requires more communication between processes.

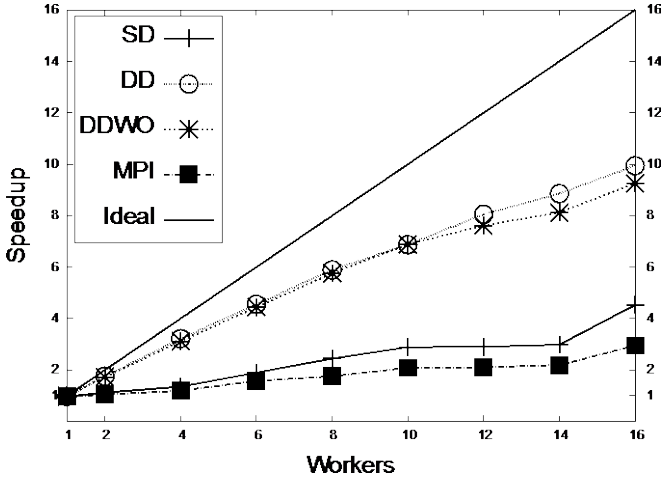
For the US 500 Airports, we observed similar results to the search engines PageRank pro- gram. The dynamic schedulers obtain very good speedups while the static ones perform badly. As usual, **MPI** follows the behavior of **SD**.



(a) All Pairs with dataset grid 35.



(b) All Pairs with dataset grid 25 (paths included).



(c) All Pairs with dataset US 500 Airports.

Figure 11: Performance results for the All Pairs Shortest Paths program. In (a) and (c) only the shortest distance is computed, while in (b) both the minimal distance and the corresponding path are computed.

5 CONCLUSION

Over the course of this project we have sucessfully ported Meld from its original use as a programming language for Claytronics and other physically embedded massively distributed systems to more traditional parallel machines including multi-core machines and those that support MPI. The exploration of Meld as a parallel programming language is still ongoing. However, it already shows promise for creating concise and understandable programs which run efficiently.

In addition to the porting of Meld to more traditional parallel machines we have incorportated linear logic into the language supporting a way to manage state change in a way that will lend itself to proving programs correct. In work related to this grant we have shown that the semantics of this approach is sound and complete.

We are still experimenting with various coordination primitives and will continue to push on algorithm development particularly for graph-based aglorithms which exhibit irregular parallelism.

REFERENCES

[1] S. C. Goldstein, J. D. Campbell, and T. C. Mowry, "Programmable matter," *IEEE Computer*, vol. 38, no. 6, pp. 99–101, June 2005.

[2] M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian, "Modular self-reconfigurable robot systems [grand challenges of robotics]," *Robotics and Automation Magazine, IEEE*, vol. 14, no. 1, pp. 43–52, March 2007.

[3] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell, "A language for large ensembles of independently executing nodes," in *Proceedings of the International Conference on Logic Programming (ICLP '09)*, July 2009.

[4] J. D. Ullman, *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. New York, NY, USA: W. H. Freeman & Co., 1990.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[6] R. S. Nikhil, "A multithreaded implementation of id using p-risc graphs," in *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. London, UK: Springer-Verlag, 1994, pp. 390–405.

[7] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken, "Tam—a compiler controlled threaded abstract machine," *J. Parallel Distrib. Comput.*, vol. 18, no. 3, pp. 347–370, 1993.

[8] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson, "Report on the programming language haskell: a non-strict, purely functional language version 1.2," *SIGPLAN Not.*, vol. 27, no. 5, pp. 1–164, 1992.

[9] G. E. Blelloch and J. Greiner, "A provable time and space efficient implementation of nesl," in *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 1996, pp. 213–225.

[10] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the sisal language project," *J. Parallel Distrib. Comput.*, vol. 10, no. 4, pp. 349–366, 1990.

[11] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *International Conference on Compiler Construction*, Grenoble, France, Apr 2002. [Online]. Available: http://groups.csail.mit.edu/commit/papers/02/streamit-cc.pdf

[12] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo, "An hpf compiler for the ibm-sp2," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA. ACM, 1995, p. 71.

[13] D. D. Chamberlin and R. F. Boyce, "Sequel: A structured english query language," in *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. New York, NY, USA: ACM, 1974, pp. 249–264.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *European Conference on Computer Systems (EuroSys)*, Lisbon, Portugal, March 21-23 2007, pp. 59–72, also as technical report MSR-TR-2006-140. [Online]. Available: http://budiu.info/work/eurosys07.pdf

[15] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *Symposium on Operating System Design and Implementation (OSDI)*, San Diego, CA, December 8-10 2008, p. 14. [Online]. Available: http://budiu.info/work/DryadLINQ.pdf

[16] G. Czajkowski, "Large-scale graph computing at google," googleresearch.blogspot.com/2009/06/large-scale-graph-computing-at-google. html, June 2009.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 1099–1110.

[18] H. Gallaire and J. Minker, Eds., *Logic and Data Bases*. Perseus Publishing, 1978.

[19] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking: language, execution and optimization," in *Proc. of the 2006 ACM SIGMOD int'l conf. on Management of data*. New York, NY, USA: ACM Press, 2006, pp. 97–108.

[20] ——, "Declarative networking," *Communications of the ACM*, vol. 52, no. 11, pp. 87–95, 2009.

[21] D. Chu, A. Tavakoli, L. Popa, and J. Hellerstein, "Entirely declarative sensor network systems," in *Proceedings of the 32nd international conference on Very large data bases*, 2006, pp. 1203–6.

[22] S. Lucco and O. Sharp, "Parallel programming with coordination structures," in *POPL '91: Proceedings of the 18th ACM SIGPLAN- SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1991, pp. 197–208.

[23] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, 1992.

[24] K. M. Kahn and M. S. Miller, *Ecology of Computation*. North-Holland, 1988, ch. Language Design and Open Systems.

[25] J. Dongarra, D. Sorenson, and P. Brewer, *Aspects of Computation on Asynchronous Processors*. North-Holland, 1988, ch. Tools and methodology for programming parallel processors, pp. 125–138.
[26] S. C. Goldstein, T. C. Mowry, J. D. Campbell, M. P. Ashley-Rollman, M. D. Rosa, S. Funiak, J. F. Hoburg, M. E. Karagozler, B. Kirby, P. Lee, P. Pillai, J. R. Reid, D. D. Stancil, and M. P. Weller, "Beyond audio and video: Using claytronics to enable pario," *AI Magazine*, vol. 30, no. 2, July 2009.
[27] C. Group, "Claytronics project pages," www.cs.cmu.edu/ ~claytronics, 2009.
[28] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS '07)*, October 2007. [Online]. Available: http://www.cs.cmu.edu/ ~claytronics/papers/ashley-rollman-iros07.pdf
[29] D. Dewey, S. S. Srinivasa, M. P. Ashley-Rollman, M. De Rosa, P. Pillai, T. C. Mowry, J. D. Campbell, and S. C. Goldstein, "Generalizing metamodules to simplify planning in modular robotic systems," in *Proceedings of IEEE/RSJ 2008 International Conference on Intelligent Robots and Systems IROS '08*, Nice, France, September 2008. [Online]. Available: http://www.cs.cmu.edu/ ~claytronics/papers/dewey-iros08.pdf
[30] M. P. Ashley-Rollman and P. Pillai, "Claytronics - gates hillman complex," 2009. [Online]. Available: www.youtube.com/watch?v= Vu6RnR-0Dtw
[31] C. Zaniolo, N. Arni, and K. Ong, "Negation and aggregates in recursive rules: the ldl++ approach," in *Deductive and Object-Oriented Databases*, 1993, pp. 204–221.
[32] J.-Y. Girard, "Linear logic," *Theor. Comp. Sci.*, vol. 50, pp. 1–102, 1987.
[33] P. López, F. Pfenning, J. Polakow, and K. Watkins, "Monadic concurrent linear logic programming," in *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, A.Felty, Ed. Lisbon, Portugal: ACM Press, July 2005, pp. 35–46.
[34] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking: language, execution and optimization," in *Proc. of the 2006 ACM SIGMOD int'l conf. on Management of data*. New York, NY, USA: ACM Press, 2006, pp. 97–108.
[35] B. Lubachevsky and D. Mitra, "A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius," *J. ACM*, vol. 33, pp. 130–150, January 1986. [Online]. Available: http://doi.acm.org/10.1145/4904.4801
[36] J. Gonzalez, Y. Low, and C. Guestrin, "Residual splash for optimally parallelizing belief propagation," in *In Artificial Intelligence and Statistics (AISTATS)*, Clearwater Beach, Florida, April 2009.
[37] I. Balbin and K. Ramamohanarao, "A generalization of the differential approach to recursive query evaluation," *The Journal of Logic Programming*, vol. 4, no. 3, pp. 259 – 262, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0743106687900045
[38] F. Bancilhon, "Naive evaluation of recursively defined relations," in *On Knowledge Base Management Systems (Islamorada)*. New York, NY, USA: Springer-Verlag New York, Inc., 1985, pp. 165–178.
[39] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, December 1998. [Online]. Available: http://dx.doi.org/10.1137/S1064827595287997
[40] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang, "Parallel garbage collection for shared memory multiprocessors," in *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, ser. JVM'01. Berkeley, CA, USA: USENIX Association, 2001, pp. 21–21. [Online]. Available: http://portal.acm.org/citation.cfm?id=1267847.1267868
[41] E. W. Dijkstra, "Shmuel Safra's version of termination detection," Jan. 1987, unpublished note. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF
[42] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms." in *PODC'96*, 1996, pp. 267–275.
[43] N. Shavit and A. Zemach, "Diffracting trees," in *In Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures. ACM*, 1994.
[44] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren, "Efficient Tabling Mechanisms for Logic Programs," in *International Conference on Logic Programming*. The MIT Press, 1995, pp. 687–711.
[45] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, "Distributing hot-spot addressing in large-scale multiprocessors," in *Interconnection networks for high-performance parallel computers*, I. D. Scherson and A. S. Youssef, Eds. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 302–309. [Online]. Available: http://portal.acm.org/citation.cfm?id=201173.201223
[46] Datalog is related to Prolog, but restricts usage (one important restriction is that it is forward-chaining) so it is sound and complete.
[47] It has been shown in [31] that set aggregates can be implemented using negation, however we implement aggregates directly, without using source code transformation.
[48] All programs are accessible here: https://github.com/flavioc/meld/tree/master/benchs/progs/sources.
[49] Obtained from http:toreopsahl.com/datasets/
[50] Obtained from http://www.cs.toronto.edu/ ~tsap/experiments/download/download.html

APPENDIX

Here we list the source code for the Meld programs we evaluated.

A.1 All Pairs

This program computes the shortest distance and the corresponding path between all pairs of nodes in a graph.

```
type path(node, node, min int, list node).

path(A, A, 0, [A]).

path(A, B, D + W, [A | P]) :-
edge(A, C, W),
path(C, B, D, P).
```

A.2 Pagerank

Pagerank computes the rank of a node.

```
type pagerank(node, int, float).
type calcRank(node, int, sum float [<- edge]).
type numLinks(node, sum int).

const damping = 0.85.
const iterations = 35.
const pages = @world.

numLinks(A, 0).
numLinks(A, 1) :- edge(A, B).

pagerank(A, 0, 1.0 / float(pages)).
pagerank(A, I, V) :-
    I > 0,
    calcRank(A, I, T),
    V = damping + (1.0 -
    damping) * T, I <=
    iterations.

 calcRank(A, I + 1, 0.0) :-
    pagerank(A, I, _).
 calcRank(A, I + 1, 0 / float(C)) :-
    edge(B, A),
    pagerank(B, I, O),
    numLinks(B, C).
```

A.3 Neural Network

The Neural Network program implements the training phase of the back-propagation algorithm for neural networks.

```
type activated(node, int, float).
type expected(node, int, float).
type receive(node, int, sum float [<- link]).
type hiddengrad(node, int, float).
type allhiddengrad(node, int, sum float [-> link]).
type outgrad(node, int, float).
type delta(node, int, node, float).

extern float sigmoid(float).

const lrate = 0.01.

receive(A, I, W * V) :-
    link(B, A),
    weight(B, I, A, W),
    activated(B, I, V).

activated(A, I, sigmoid(V)) :-
    receive(A, I, V).

delta(A, I, B, lrate * V * G) :-
    hidden(A),
    activated(A, I, V),
    link(A, B),
    outgrad(B, I, G).

outgrad(A, I, G * (1.0 - G) * (E - G)) :-
    expected(A, I, E),
    activated(A, I, G).

allhiddengrad(A, I, V * W) :-
    hidden(A),
    link(A, B),
    weight(A, I, B, W),
    outgrad(B, I, V).

hiddengrad(A, I, V * (1.0 - V) * T) :-
    activated(A, I, V),
    allhiddengrad(A, I, T).

delta(A, I, B, lrate * V * G) :-
    input(A),
    activated(A, I, V),
```

```
    link(A, B),
    hiddengrad(B, I, G).

weight(A,I + 1, B, 0 + D) :-
weight(A, I, B, O),
delta(A, I, B, D).
```

A.4 Belief Propagation

This programs implements the naive loopy belief propagation algorithm.

```
type route edge(node, node).
type coord(node, int).
type potential(node, list float).
type belief(node, int, list float).
type globalpotential(node, list float).
type message(node, int, node, list float).

type beliefmul(node, int, sum list float [-> edge]).

extern list float normalize(list float).
extern list float damp(list float, list float, float).
extern list float divide(list float, list float).
extern list float convolve(list float, list float).
extern list float addfloatlists(list float, list float).

const iterations = 35.
const damping = 0.1.

beliefmul(A,I + 1, L) :- edge(A, B), message(B, I, A, L).

belief(A, I, normalize(addfloatlists(L, P))) :-
    beliefmul(A, I, L), potential(A,P), I <= iterations.

message(A, I + 1, B, damp(Convolved,OldOut, damping)) :-
    I <= iterations,
    belief(A, NI, Belief),
    globalpotential(A, GP),
    NI = I + 1,
    edge(A,B),
    message(B, I, A, OldIn),
    message(A, I, B, OldOut),
    Cavity = normalize(divide(Belief, OldIn)),
    Convolved = normalize(convolve(GP, Cavity)).
```

A.5 Power Grid

```
type route edge(node, node).
type source(node).
type sink(node, int).
type linear pluggedIn(node, node).
type linear unplugged(node).
type linear load(node, float).

unplugged(Sink) :- sink(Sink, _).
load(Source, 0) :- source(Source).

pluggedIn(Sink, Source),
load(Source, OldAmt + Amt) :-
   unplugged(Sink),
   !sink(Sink, Amt),
   !edge(Sink, Source),
   !source(Source), load(Source, OldAmt).

pluggedIn(Sink, NewSource),
load(OldSource, OldSourceAmt - Amt),
load(NewSource, NewSourceAmt + Amt) :-
pluggedIn(Sink, OldSource),
!edge(Sink, NewSource),
load(OldSource, OldSourceAmt),
OldSourceAmt > 1,
load(NewSource, NewSourceAmt),
!sink(Sink, Amt),
NewSourceAmt + Amt < OldSourceAmt || rand().

terminate() :- proved(pluggedIn) > MAX_ITERATIONS.
```